Glance – Generalized Geometric Primitives and Transformations for Information Visualization in AR/VR Environments

Daniel Filonik* Tomasz Bednarz† Markus Rittenbruch‡ Marcus Foth§
Urban Informatics Research Lab*‡§ | Institute for Future Environments*‡, Queensland University of Technology
ARC Centre of Excellence for Mathematical & Statistical Frontiers† | CSIRO Data61†

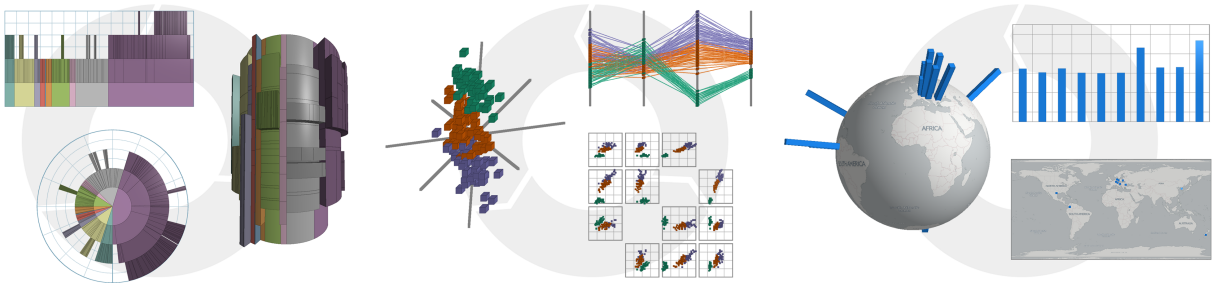

Figure 1: *Interactively exploring multidimensional, multivariate data with dynamic visual representations.*

Abstract

This paper outlines *Glance*, a unifying framework for exploring multidimensional, multivariate data in the context of *AR/VR* environments, along with specific implementation techniques that utilize programmable GPUs. The presented techniques extend the graphics pipeline through programmable shaders in order to support more general geometries and operations. Our point of departure from existing structural theories of graphics is a general *spatial substrate*, where data is encoded using higher-dimensional *geometric primitives*. From there, we define a series of processing stages, utilizing shaders to enable flexible and dynamic *coordinate transformations*. Furthermore, we describe how advanced visualization techniques, such as faceting and multiple views, can be integrated elegantly into our model. Bridging between *Computer Graphics* and *Information Visualization* theories, the elements of our framework are composable and expressive, allowing a diverse set of visualizations to be specified in a universal manner (see figure 1).

CR Categories: I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction techniques; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Hierarchy and geometric transformations

Keywords: GPU-acceleration, shaders, interaction, visualization

Links: DL PDF VIDEO CODE

*e-mail:daniel.filonik@qut.edu.au
†e-mail:tomasz.bednarz@qut.edu.au
‡e-mail:m.rittenbruch@qut.edu.au
§e-mail:m.foth@qut.edu.au

1 Introduction

Conventional computer graphics are optimized for displaying 2- or 3-dimensional scenes, often motivated by various forms of entertainment or simulation, which aim to model our perception of reality as closely as possible. However, in the context of *Information Visualization*, analysts work with abstract data and are not limited by the constraints of realism. Furthermore, the data sets encountered in this context may have challenging characteristics, such as large numbers of attributes which cannot be represented in merely three spatial dimensions. In order to cope with such complex data sets, researchers have devised highly diverse visualization techniques, as surveyed by Wong and Bergeron [1994]. Through carefully chosen projections or creatively reconfigured spatial arrangements, coupled with corresponding interaction mechanisms, such techniques can produce unusual and compelling visual representations. While the topics covered in this paper are founded in well-established mathematics, they are still relatively unexplored in the area of real-time graphics. Although a lot of mathematical knowledge informs the domains of *Exploratory Data Analysis* and *Information Visualization*, further formal treatment can foster quality and rigor in the practice of visualization, resulting in more precise descriptions and resolving conceptual ambiguities. One reason why such a treatment is still lacking at times, is because it is often not trivial to apply these mathematical concepts in practical implementations. This article emphasizes connections between applied mathematics and visualization theories, and provides practical recipes for applying these ideas and concepts. Although our work draws from extensive subjects like *Geometric Algebra* and *Tensor Calculus*, we limit the discussion to key concepts that are relevant to the implementation.

On a high level, the approach presented throughout this article is one of generalization and extension of the traditional graphics pipeline through programmable shaders in order to gain expressiveness. The benefit of this approach is that it allows us to consider seemingly disparate visualization types in a unified framework. Furthermore, since the model is based on geometric primitives and transformations in generalized spatial substrate, it preserves the full dimensionality of the data throughout the graphics pipeline until the final projection step. Similarly, the model makes no assumptions about the dimension of the final display space, allowing the resulting visualizations to be seamlessly integrated in

AR/VR environments and enabling natural interactions in such contexts. Our discussion covers GPU-based techniques for efficiently implementing a wider range of geometric primitives and transformations than commonly encountered in conventional computer graphics. In recent years, there has been a clear trend towards more general-purpose computation on GPUs, which lays the groundwork to enable the proposed techniques. However, despite this trend, our work also exposes some limitations in current computer graphics APIs and hardware. These limitations are not surprising, since the requirements for some of the proposed techniques can be unusual compared to conventional applications. Nevertheless, we hope that by demonstrating their usefulness in the domain of *Information Visualization*, we will encourage future graphics hardware and APIs to lift further restrictions and expand these capabilities.

The code listings in this article are given in the *OpenGL Shading Language*. The code was tested using the current OpenGL 4.5 [Khronos Group 2015], but most techniques are compatible with versions as far back as 3.2.

2 Related Work

Beginning with the work of Bertin [1983], the systematic study of graphical representations has resulted in an important set of visualization theories, collectively referred to as structural theories of graphics. Early on, Mackinlay [1986] built and extended upon these fundamental components and structures of these theories in order to develop automated presentation tools. The idea of deconstructing visualizations into fine-grained, modular units of composition also lies at the heart of graphics grammars [Wilkinson 2005; Wickham 2010]. These theories provide the foundation for visualization interfaces developed as part of ongoing research, such as *Lyra* [Satyanarayan and Heer 2014], as well as commercial software, such as *Tableau* and its predecessor *Polaris* [Stolte et al. 2002]. Inspired by expressive power of these grammatical descriptions, we set out to develop a GPU-accelerated implementation, resulting in the techniques presented in this article.

We adopt core ideas and terminology from multiple approaches, organizing them in a way that is natural within the context of our implementation. Predominantly, our terminology is based on the concepts introduced by Card et al. [1999], breaking down visual structures into *spatial substrate*, *graphical marks*, *connections and enclosure*, *retinal properties*, and *temporal encoding*. Corresponding concepts can be found in graphics grammars, alongside others that have no direct counterpart. The notions of *spatial substrates* were already explored by Bertin [1983], who focused on the 2-dimensional plane as a substrate. Other researchers have extended these ideas, such as Mackinlay [1986], whose system allowed for 1- to 3-dimensional substrates. Our approach adopts a fully generalized n-dimensional substrate as a starting point. This makes it possible to avoid ambiguities and inconsistencies of previous models, especially in the context of visualization techniques for multidimensional, multivariate data, such as parallel coordinates or parallel sets. Furthermore, taking inspiration from computer graphics, we have found it advantageous to collapse the aspects of *graphical marks*, *connections and enclosure*, and *retinal properties* into a versatile model of *geometric primitives*, rather than viewing them in isolation. Finally, a key characteristic of our approach is to provide great flexibility with regard to *coordinate transformations*, which play an integral role in graphics grammars, as examined by Wilkinson [2005].

A key motivation for GPU-acceleration is its ability to enable highly dynamic and interactive visualizations. Given the growing volume and complexity of data, specialized graphics hardware is going to remain critical to achieve fluent, uninterrupted user experiences, which Heer and Shneiderman [2012] characterize as visual analytics resonating with the pace of human thought. Liu and Heer [2014] have found that latency in interactive analysis tools negatively impacts the results of exploratory visual analysis. Therefore, interactive performance should be regarded as a key requirement of visualization software rather than being merely optional. Going further, the computational power of GPUs along with novel interaction modalities opens up entirely new avenues for analysis. For example, [Brosz et al. 2013] demonstrate a touch interface to casually perform sophisticated graphical deformations. Other documented benefits of real-time interactivity include more natural representation of temporal changes in data, as well as better spatial perception, using motion to recover information lost during projection [Weiskopf 2007]. As stated by Kosara [2003], a comprehensive mental image of multidimensional, multivariate data is only formed by means of user interaction. An ongoing area of research is the use of animation to improve interaction and understanding. Heer and Robertson [2007] have studied animated transitions between visualization and found that they can significantly improve graphical perception. Instant and continuous feedback establishes relationships between graphical elements in changing views, helping analysts to retain context. We believe that paired with the expressive power of graphics grammars, this holds the promise of truly free-flowing exploration through incremental and iterative specification of graphical representations. In a similar vein, Ruchikachorn and Mueller [2015] propose the use of animations as a method of teaching unfamiliar visual representations to visualization novices. The techniques discussed in this article are well suited for implementing such animated transitions.

The programmable pipeline of modern graphics cards provides multiple potential entry points for enhancing visualizations. On the one hand, *image-based* techniques operate on the final stages of the pipeline, applying image processing operations to fragments. Image-based techniques work with rasterized representations in image space. While these operations may take into account additional information from other fragment buffers – including depth or motion – working with a rasterized representation inevitably loses information about the scene geometry, resulting in potential artifacts and limitations. On the other hand, *geometry-based* techniques operate on earlier stages of the pipeline, processing and transforming geometric primitives. Such operations have access to the full geometric information of the scene. McDonnel and Elmqvist [2009] have recognized the under-utilization of GPUs in *Information Visualization*, when compared to *Scientific Visualization*. In response, the authors propose a formal model for image space visualization operations, along with a visual programming environment for creating shaders. Aside from enabling visualization-specific image processing filters, warping, and distortions, this model even supports generating graphical marks in image space. Still, their approach is strictly limited to *image-based* techniques. In contrast to that, Florek and Novotnỳ [2006] provide an example of a *geometry-based* technique, using the geometry processing stages of the graphics pipeline to produce an interactive parallel coordinates display. Furthermore, a series of articles by Bailey [2009; 2011; 2013] covers a selection of specific GPU techniques, both *image-* and *geometry-based*. Our approach resembles the latter examples, however the techniques that we propose are more general and not limited to specific visual representations.

Heckbert [1994] and Salomon [2012] contain detailed instructions on generalizing the core concepts of computer graphics to higher dimensions. [Chu et al. 2009] describe GL4D, a rendering architecture and implementation that exploits programmable shaders to perform high-quality interactive 4D rendering and visualization. Compared to our work, GL4D is a much more specialized 4D rendering solution, including full 4D lighting, culling, and transparency.

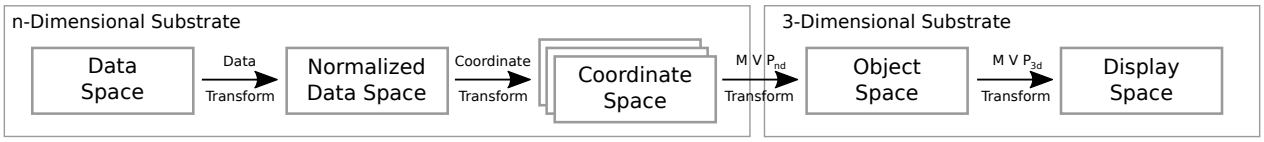

Figure 2: *Transformation stages of the proposed extended graphics pipeline.*

However, while direct extensions of rendering techniques may be suitable for investigating higher-dimensional manifolds, there are often better representational choices for visualizing abstract information. In contrast, our approach omits more advanced graphics functionality, focusing instead on a minimal feature set needed for visualization of multidimensional, multivariate data sets.

3 Method

Our approach towards visualization throughout this article is succinctly captured by McCormick et al. [1987], who state that:

> "Visualization is a method of computing. It transforms the symbolic into the geometric."

In other words, geometry is the mathematical link between abstract information and graphic representation. In order to visually represent information, we assign to it a geometric form.

In the following sections, we outline how we generate and transform geometries to visually encode abstract data provided by our users. We start with an overview of the extended graphics pipeline used in our visualization system in section 3.1. Subsequently, we detail the implementation technique used to work with higher dimensional spatial substrates on GPU in section 3.2. An example of a simple, yet versatile model of geometric primitives for generating graphical marks is discussed in section 3.3. Finally, we discuss techniques allowing user to interactively transform and modify the visualization, for example by deforming or reconfiguring the axes in order to achieve a diverse range of graphical representations, as detailed in sections 3.4 and 3.5.

3.1 Graphics Pipeline

In order to be practical for AR/VR environments, our techniques for rendering visualizations were heavily shaped by performance considerations. For one, a proven strategy for achieving high-performance in interactive graphics is to minimize copies of data between CPU- and GPU-memory. Further, it is also advisable to minimize the number of draw calls, which also cause synchronization between the CPU and the GPU. Therefore, we employ a graphics programming technique commonly referred to as geometry instancing, which is frequently employed to render complex particle systems. Several variations of this technique exist, which are applicable depending on the specific situation. For simple geometries, our approach relies on geometry shaders to generate marks entirely on the fly. For more complex geometries, we employ the native instancing functionality provided by OpenGL, as covered in [Ginsburg et al. 2014].

The programmable graphics pipeline consists of several processing stages, during which the initial geometry typically undergoes several transformation steps, most commonly broken down into *model*, *view*, and *projection* (MVP). However, in more general terms, it may be necessary to perform multiple iterations of transformations and projections, especially when working with higher dimensional geometries and non-linear transformations. Rather than going to such a full level of generality, we introduce a set of purposefully chosen steps at the beginning of the conventional pipeline, in order to address the needs of *Information Visualization*. The process begins with data points in a general n-dimensional spatial substrate, which are transformed and projected – generating mark geometries along the way at the appropriate transformation step – to ultimately produce a 2- or 3-dimensional geometric representation.

The abstract data in *Information Visualization* can be categorized in a number of different ways. A common distinction is between *categorical* and *quantitative*, which may respectively be further divided into *nominal* and *ordinal*, as well as *discrete* and *continuous*. We recognize the importance of these distinctions, and believe that they are valuable in the visual encoding process. However, ultimately the vertex structures on the GPU only contain geometric information relevant for display. Therefore, we assume that both, *categorical* and *quantitative* data will be mapped to numerical coordinate values in our vectors, which always contain floating point numbers. The first transformation maps the input data points to a unit hypercube in our general spatial substrate, normalizing the coordinates of visible marks to lie within the interval $[-1; +1]$. This typically involves translation and scaling. Analogous to *normalized device space*, we refer to this as *normalized data space*, and the main purpose of this transformation is to perform clipping. The subsequent stage performs coordinate transformations – one or more of which can be applied sequentially or in parallel (to generate multiple views) – and the resulting in vertices are considered to lie in the chosen *coordinate space*. The final stage transforms and projects the n-dimensional geometries from *coordinate space* into *object space*. Following the dimension reduction, the geometries may pass through a conventional series of transformations in order to be placed within the scene. The dimension reduction is typically where information loss occurs, however more sophisticated visualization techniques can be used to compensate for this issue.

We illustrate the process using two examples (see figures 3 and 4). First, we use wind data collected by U.S. Geological Survey. The data set contains periodic wind measurements aggregated over several months. We encode the data attributes direction, frequency, and date in our spatial substrate. As our goal is to generate a *wind rose*, we clip marks that lie outside the chosen date range, and subsequently project to 2-dimensional polar coordinate space. Therefore, the initial *data space* is 3-dimensional, whereas the final geometric representation in *display space* is 2-dimensional. However, since all of the data is readily available in GPU-memory, interactive functionality can easily be added to allow the users to rapidly navigate through months by merely adjusting the data transform – effectively recovering the third dimension.

Second, we use Fisher's Iris data set [Fisher 1936]. This commonly encountered exemplary data set is based on measurements collected from three species of Iris, comprising four plant features, as well as a class label. The four measurements can be directly encoded as positions in a 4-dimensional spatial substrate, whereas a qualitative color palette is used to assign colors to the class labels. As our goal is to generate a *scatterplot matrix*, we project the marks from normalized data space to multiple cartesian coordinate spaces, one for each combinations of axes. Finally, the spaces are transformed into the desired matrix arrangement.

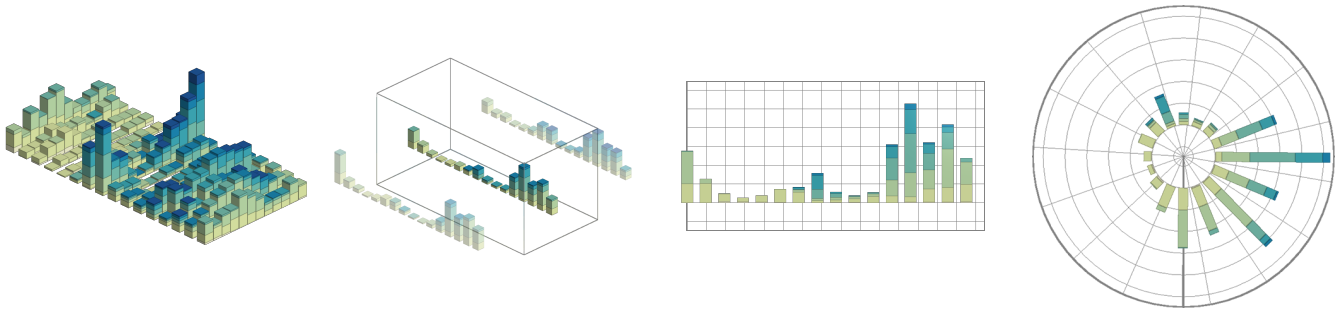

Figure 3: *Wind data normalized and clipped against 3-cube, projected and transformed to polar coordinate space.*

While different transformations significantly alter the resulting visual appearance, it is worth highlighting that the images in each example were obtained from a single underlying visualization specification with successively more transformation steps applied. This is one of the key insights for unifying seemingly disparate visualization types, and allowing users to explore them interactively with smoothly interpolated transitions. During initialization, the raw data is transformed according to the underlying visualization specification to populate the vertex buffer objects, whereas the final geometric representation is dynamically calculated on the fly. This illustrates another basic principle of our approach – moving data into vertex buffer objects as early on as possible and leveraging the GPU to perform further processing.

3.2 Spatial Substrate

Before it is possible to specify any geometries to represent data, it is first necessary to define the *spatial substrate* that they occupy. The relationship between geometric space and objects embedded within it is intricate, and will be discussed throughout the following sections. In linear algebra, the abstract concept of a vector space is used to formally describe spatial relationships. Accordingly, the abstract elements of computation are referred to as vectors. A vector over real numbers $\vec{v} \in \mathbb{R}^n$ can be described by its coordinate components $v_i \in \mathbb{R}$ for $i = 1 \ldots n$. Throughout this article we aim to express geometric computations in a coordinate-free manner, specifying them in terms of operations on vectors, rather than their coordinates. As a result, computation specified in a coordinate-free manner trivially extend to vector spaces of higher dimensions.

Conventional computer graphics model 3-dimensional scenes by embedding the base space $\mathbb{R}^3$ in a homogeneous representation space $\mathbb{R}^{3+1}$, which allows for expressing most common geometric operations as linear transformations using matrices. For this purpose, shading languages provide 4-vectors and 4×4-matrices, with highly optimized arithmetic operations. When generalizing this concept to higher dimensional base spaces $\mathbb{R}^n$ with homogeneous representation spaces $\mathbb{R}^{n+1}$, our goal is to utilize the existing capabilities. Fortunately, we can exploit a property of matrices, which allows us to split larger matrices into smaller chunks and define operations in terms of operations on said chunks. Such matrices are called *partitioned*, or *block matrices*, and their properties have been well studied [Eves 1980]. Therefore, in order to process higher-dimensional data on the GPU, we introduce preprocessor definitions for arrays of vectors and matrices, where the number of elements can be configured via preprocessor constant. Subsequently, we define functions to express all major arithmetic operations in terms of efficient operations on 4-vectors and 4×4-matrices, as illustrated by the dot product in listing 1. Although implementation dependent, it is safe to assume that the necessary loops will be fully unrolled by the GLSL compiler.

```
#define vecN vec4[CHUNK_COUNT]
#define matN mat4[CHUNK_COUNT][CHUNK_COUNT]

vecN dotN(matN lhs, vecN rhs) {
    vecN result = zerosN();
    for(int i = 0; i<CHUNK_COUNT; i++) {
        for(int j = 0; j<CHUNK_COUNT; j++) {
            result[j] += lhs[i][j] * rhs[i];
        }
    }
    return result;
}
```

Listing 1: *Dot product with chunked matrix and vector.*

Aside from the n-dimensional base space, another useful notion is that of k-dimensional subspaces with $0 <= k <= n$. A k-dimensional subspace is spanned by a set of k independent vectors in $\mathbb{R}^n$, which form a basis of the subspace. Therefore, the dimension of the base space provides an upper limit for the dimension of the subspaces spanned within. In other words, we may choose an arbitrary set of k independent vectors in order to specify a basis of a *coordinate system* within our *spatial substrate*, which is used to provide users with the capability to interactively add and remove *coordinate axes* as needed when exploring data. Furthermore, for every k-dimensional space in an n-dimensional base space, there exists a (n-k)-dimensional dual space, also refereed to as the orthogonal complement. The notion of orthogonal complement is common in 3D graphics with respect to planes (2D subspaces) and their normals (1D subspaces). In the more general form, it is key to our definition of *geometric primitives*, as discussed in 3.3.

While the larger vectors and matrices inevitably become computationally expensive, the GPU's ability to perform large numbers of floating point operations makes it possible to achieve interactive performance with thousands of higher dimensional data points on a standard consumer hardware[1]. However, there are limitations to this technique. The amount of data that can be supplied for a given vertex is limited by the maximum number of possible attributes (`GL_MAX_VERTEX_ATTRIBS`), which is determined by graphics hardware and driver. Commonly, this number is 16, which means that for vertices only containing positions, it is possible to have up to 64-dimensional base spaces. However, typically it is necessary to introduce further vertex attributes as discussed in the following section, therefore the number can be a limiting factor. Another potential pitfall is the memory layout of the chunked matrices, which may be incompatible with common linear algebra libraries on the CPU-side. In this case it is necessary to convert the matrices when transferring to the GPU-side. Since matrices grow large, it makes sense to store them in uniform buffer objects.

[1] Apple Macbook Pro, Intel Core i7-3635QM @ 2.40GHz, NVIDIA GeForce GT 650M

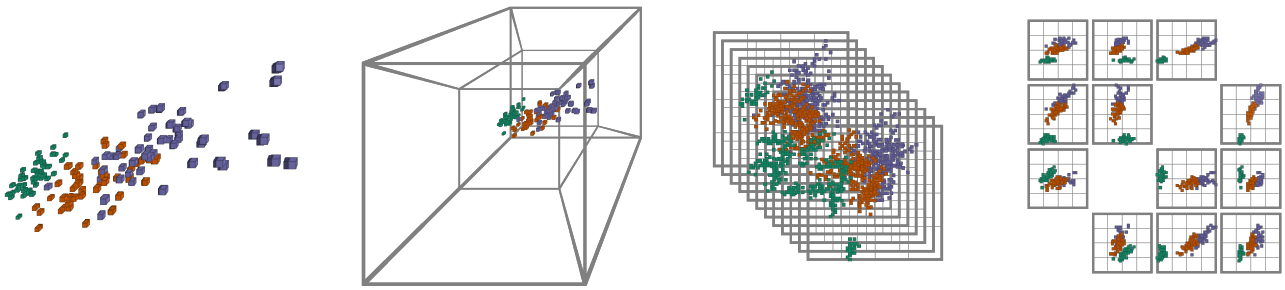

Figure 4: *Iris data normalized and clipped against 4-cube, projected and transformed to multiple cartesian coordinate spaces.*

3.3 Geometric Primitives

Our mark model is based on the *retinal variables* identified by Bertin [1983], translated into the context of modern computer graphics and expressed in a generalized fashion. In particular, the final geometry of our graphical marks is affected by the variables of *position*, *attitude*, and *size*. In addition to the geometric information, our vertex structure also contains members representing the variables of *color (hue, saturation, value)*, *pattern* and *shape*. The vertex attributes store information that varies between individual marks. In contrast to that, program uniforms are used to to represent information that is constant across all marks. For example, we may vary the relative sizes of individual marks to encode information, while globally controlling their minimum and maximum sizes through uniforms. For the sake of generality, our proposed implementation technique leverages the n-dimensional vector definitions described in the previous section, as shown in listing 2.

```
uniform sampler2D pattern_atlas;
uniform sampler2D shape_atlas;

uniform matN basis;
uniform matN dual_basis;

struct VertexN {
    vecN position;
    vecN attitude;
    vecN size;
    vec4 color;
    vec4 textures; // Texture Atlas Offsets
};
```

Listing 2: *Uniforms and vertex structure for mark model.*

Generally speaking, every mark occupies a bounded, continuous region in the n-dimensional spatial substrate. The extent of this region is determined by a given geometry. In real-time graphics, arbitrary geometries are conventionally approximated through polytopes. Although it is occasionally perpetrated that modern graphics APIs only process triangles, that is not technically true. While the triangle is the default 2-dimensional polytope, OpenGL supports not only 2-dimensional (`GL_TRIANGLES`), but also 1-dimensional (`GL_LINES`) and 0-dimensional (`GL_POINTS`) primitives. Before the introduction of geometry shaders, these primitives were of limited use, since there were only rudimentary ways of adjusting their appearance. However, with the advent of geometry shaders, which introduced the capability to generate new geometric primitives on the GPU, they have become invaluable for visualization purposes. Specifically, it is possible to send abstract vertices to the GPU, using their attributes as parameters to generate concrete geometries, which effectively achieves a form of model-view separation. This is used extensively by our proposed model to create the necessary marks on the fly.

In principle, arbitrary geometries can be used as marks – and a variety of glyphs have been explored for visualization purposes. In the following, we describe a simple, yet versatile geometric model, which implements marks as higher-dimensional cube geometries. Any differences in appearance between individual marks are determined by their respective sets of vertex attributes. For the most part, these vertex attributes constitute an additional transformation, with the translation provided as *position*, the stance or direction given by *attitude*, and the scaling determined by the *size*. This transformation is applied to the concrete geometry that is instanced for each mark. However, it is important to highlight that despite their non-trivial geometries, in our proposed model these marks are still conceptually considered 0^{th}-order primitives – and from the perspective of the graphics API they are supplied in the form of a single vertex (0-simplices). More generally, n^{th}-order primitives (n-simplices) can be used to visually establish relationships between $n+1$ marks. For example, the vertices of a graph would be 0^{th}-order primitives, while the edges connecting two vertices would be 1^{st}-order primitives. Customarily, graphs with edges that connect varying numbers of vertices are also referred to as hyper-graphs.

As previously indicated, a particularly useful mark geometry are axis-aligned, unit-sized hyper-cubes, which correspond to the unit volume elements of k-dimensional subspaces. For example, in a 2-dimensional base space, they can either be points (0-cubes), line segments (1-cubes), or plane segments (2-cubes). Axis-aligned hyper-cubes are exceptional in their simplicity, because they are the only n-dimensional geometries that can be constructed entirely from cartesian products of 1-dimensional geometries. Conversely, any such hyper-cube can be factored into 1-dimensional geometries by projection onto the 1-dimensional subspaces corresponding to the coordinate axes. Therefore, in terms of input data, the visualization kernel only needs to support two fundamental types, namely instants ($\mathbb{R}$) and intervals ($\mathbb{R}^2$). All subsequent geometric primitives result from cartesian products of these fundamental types. The type information about individual factors in the cartesian products is encoded within the basis uniforms, which contain a subspace basis and a dual subspace basis. The matrices are complementary in the sense that their sum equals the identity matrix. The bases are used to extrude the mark geometries in two independent steps at deliberately chosen transformation stages in the pipeline. This ensures that marks transform correctly and can be perceived properly in the final *display space*. In general, we deposit that geometric marks should have a non-zero hyper-volume, such that they do not vanish under projections. For example, on a 2D display every mark should occupy a non-zero area, whereas on a 3D display every mark should occupy a non-zero volume. Therefore, when working with k-dimensional primitives in an n-dimensional space, they should be extruded in their (n-k)-dimensional dual space to gain thickness. Conveniently, this dual space provides additional freedom for encoding information through varying thickness.

The separation into basis and dual basis captures subtleties in the interplay between geometric spaces and objects. It relates to Bertin's notion of *imposition* and *implantation*. These terms were used to describe the relationship between marks and the underlying space. From cursory observation, in 2-dimensional space, a rectangle that represents a point (0-cube) may appear indistinguishable from a rectangle that represents an area (2-cube). However, the difference becomes evident when the visualization is transformed. This relates to the fact that the dual space behaves differently under transformations than the base space. For instance, as the user zooms the view, the size of the point remains constant, whereas the size of the area changes as the underlying space expands. This effect can be achieved by using geometry shaders to first perform the extrusion along the basis in the *data space*, while delaying the extrusion along the appropriately transformed dual basis until later in the *coordinate space*.

It is worth noting that geometric primitives are useful for characterizing different visualization types. In two dimensions, *scatterplots* use primitives of type $\mathbb{R} \times \mathbb{R}$, *bar charts* may use $\mathbb{R} \times \mathbb{R}^2$ or $\mathbb{R}^2 \times \mathbb{R}$ depending on orientation, whereas *treemaps* would be given in terms of $\mathbb{R}^2 \times \mathbb{R}^2$. However, the type of primitive is not sufficient to uniquely identify a visualization type, as illustrated by the fact that one and the same primitive may be used to create *bar charts*, *stacked bar charts*, or *gantt charts* – among others. In such cases, the visualization types are inherently related and the final visual appearance depends solely on the specific data that is provided.

Finally, the *retinal variables* of shape and pattern require special consideration. Shape coding varies the external form of a mark, whereas pattern coding determines its internal texture. Both properties are well-suited for representing categorical data. While varying shapes can be achieved with actual geometries, for efficiency reasons it is often preferable to reuse the same geometry and apply different textures – especially when the shapes are complex. Whenever applicable, our implementation allows for this via texture atlases, which are indexed by each mark. The pattern atlas holds color information, whereas the shape atlas contains alpha masks.

3.4 Coordinate Transformations

The prevalent transformations in conventional computer graphics applications are rigid body motions, which are a subset of linear transformations and can therefore be represented using matrices. We gain expressiveness for visualization purposes by allowing a broader range of transformations. This carries the benefit of being able to express geometries in different *coordinate systems*, such as polar coordinates in figure 3. As Wilkinson [2005] states, changes of coordinate systems can simplify visual representations, as well as reshape graphics in order to emphasize salient features. Furthermore, such non-linear transformations may be used to distort the visualization space, revealing details or focusing on local regions.

A characteristic property of linear transformations is that they preserve straight lines. Unfortunately, for non-linear transformations this property will not always hold. A general transformation maps each point in a source space to a new point in a destination space. As previously discussed, our marks occupy continuous, bounded regions, which conceptually contain infinitely many points. While it is not feasible to individually transform infinitely many points, we can arbitrarily approximate the result of a non-linear transformation by subdividing our geometric primitives into sufficiently small pieces. Conveniently, modern graphics processors provide a dedicated solution to this problem, in the form of hardware tessellation. Therefore, our model utilizes tessellation shaders to subdivide the geometric primitives before applying any non-linear transformations in the geometry shader.

In terms of coordinates, a non-linear transformation is determined by multiple functions, where each function takes the original components as arguments and produces one of the new components. We denote these functions as $\mathbf{f} \in (\mathbb{R}^n \to \mathbb{R})^m$, where $\vec{w} = \mathbf{f}(\vec{v})$ has coordinates $w_i = f_i(v_1, \ldots, v_n)$ for $i = 1...m$. In other words, this gives a set of m functions that – when evaluated at a given point – will produce the position of the transformed point. However, the position is only part of the story. In order to correctly transform our marks, a basis is required. For this purpose, we simply use a symbolic mathematics package to obtain the partial derivatives of the component functions, which yields the Jacobian matrix. This gives a set of $n \times m$ functions that – when evaluated at a given point – will produce a local basis at the transformed point. Furthermore, it is generally desirable to eliminate scaling by normalizing the columns of the Jacobian matrix, which may also be done symbolically.

$$\mathbf{f} = \begin{bmatrix} f_1 \\ \vdots \\ f_m \end{bmatrix} \qquad \mathbf{J} = \begin{bmatrix} \frac{\partial f_1}{\partial v_1} & \cdots & \frac{\partial f_1}{\partial v_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial v_1} & \cdots & \frac{\partial f_m}{\partial v_n} \end{bmatrix}$$

This is also called the Jacobian linearization of a non-linear system about a specific operating point. It enables us to locally approximate non-linear transformations with linear transformations. Therefore, in order to support more general coordinate transformations, we implement the functions obtained in this manner in shader code. They are evaluated for each vertex to yield a matrix representing the corresponding linear transformation in the homogeneous representation space. Ideally, we aim to provide users with the flexibility to freely choose their coordinate system, as appropriate for the specific data set in question. Therefore, our proposed technique for implementing non-linear transformations is based on shader subroutines. The coordinate system can be changed by selecting the respective subroutine. The subroutine type for general transformations, along with the declaration of a subroutine uniform, is shown in listing 3.

```
subroutine matN CoordinateTransform(vecN v);

subroutine uniform CoordinateTransform
    coordinate_transform;
```

Listing 3: *Non-linear coordinate transformations.*

Additional parameters may be introduced into the equations to support smoothly interpolated transitions between coordinate systems. A concrete example is provided in listing 4, which shows the shader code used to interpolate between cartesian and polar coordinates based on the value of the `alpha` uniform.

```
uniform float alpha = 0.0;

subroutine(CoordinateTransform)
matN coordinate_transform_polar(vecN v) {
    float r = getN(v, 0), a = getN(v, 1);
    float c = cos(a * alpha);
    float s = sin(a * alpha);
    vecN w = v;
    if (alpha > M_EPS) {
        float focus = 1.0/alpha - 1.0;
        setN(w, 0, (r + focus) * c - focus);
        setN(w, 1, (r + focus) * s);
    }
    matN J = identityN(mat2(+c, -s, +s, +c));
    return dotN(translateN(w), J);
}
```

Listing 4: *Cartesian to polar transformation.*

Setting aside dynamic shader code generation, with this technique it is only possible to select from a predefined set of subroutines at run-time. However, it is worth noting that the above recipe based on Jacobian linearizations is completely general and works for any transformation given by differentiable functions. Therefore, the visualization system can easily be extended with additional coordinate systems by defining the necessary subroutine instances. From our experience, a small number of *coordinate transformations* generalized across various dimensions – such as spherical and hyperbolic coordinates – will cover most common use cases.

3.5 Facets and Multiples

Complex data sets often calls for more sophisticated visualization approaches, often involving multiple coordinate spaces. On the one hand, *faceting* refers to partitioning the data set into groups, and drawing each group in a separate coordinate space. On the other hand, *multiples* are separate views of the same data set in different coordinate spaces. In this section, we will focus primarily on *multiples*, however the same basic implementation technique can be applied in both cases. In our approach, such results can be achieved by introducing multiple disjoint coordinate spaces.

We propose a technique that allows us to efficiently render multiple views of a data set with a single draw call. This relies on the feature of geometry shader invocations, which instructs the shader to be run multiple times. The maximum number of invocations is given by `GL_MAX_GEOMETRY_SHADER_INVOCATIONS`, and is guaranteed to be at least 32. This is the upper bound on the number of independent coordinate systems that we may generate from a single draw call, simply by providing multiple transformation matrices acting on our data points. Similarly, if we desire the flexibility to apply non-linear transformations to multiple disjoint coordinate spaces, we may use subroutine uniform arrays.

First, the geometry shader is instructed be executed multiple times using the invocations parameter, which is configured using preprocessor constant (`MULTIPLE_COUNT`). Subsequently, in each invocation of the geometry shader, we select a different transformation from the provided set of coordinate transformations and apply it to the vertices. Especially in higher dimensions, these transformation matrices take up considerable space. Therefore, if not already previously done so, it is highly advisable to store them in uniform buffers, as illustrated by the code provided in listing 5.

```
uniform matN M;  // Model
uniform matN VP; // View, Projection

uniform TransformData {
    matN transform[MULTIPLE_COUNT];
};

void generate_vertex(VertexN v) {
    gs.vertex = v;
    gl_Position = reduceN(dotN(VP, v.position));
    EmitVertex();
}

void generate_primitive() {
    matN m = dotN(M, transform[gl_InvocationID]);
    for(int i=0; i<gl_in.length(); ++i) {
        VertexN v = tes[i].vertex;
        v.position = dotN(m, v.position);
        generate_vertex(v);
    }
    EndPrimitive();
}
```

Listing 5: *Multiple transformations in single draw call.*

One consequence of using disjoint coordinate spaces is that a single mark appears multiple times in different positions. It is sometimes be useful to highlight the relationship between different instances of the same mark with connecting geometries, which we refer to as *joins*. These join geometries can be also generated with an additional draw call using a separate geometry shader. In particular, in the case of simple point marks (geometric primitive of type $\mathbb{R}^n$) the join geometry is a line strip.

A classic example of this approach are *parallel coordinates*. In our conceptual framework, *scatterplots* are closely related to *parallel coordinates*, and we can transition from the latter to the former by projecting the points from our n-dimensional substrate onto n disjoint, 1-dimensional coordinate spaces, with subsequent transformations that arrange them to be evenly spaced and parallel to each other. However, we are not limited to this specific arrangement, and we can give users the flexibility to interactively transform the coordinate axes in any way they desire, resulting in arrangements such as those explored by Claessen and Van Wijk [2011]. More generally still, we may decompose our n-dimensional substrate into $\binom{n}{k}$ disjoint, k-dimensional coordinate spaces. Another noteworthy observation is that *marimekko* or *mosaic charts* are related to *parallel sets* in very similar manner, with the main difference being that *scatterplots* use geometric primitives of type $\mathbb{R}^n$, whereas *mosaic charts* use $(\mathbb{R}^2)^n$.

4 Conclusion

In this work, we focus on extensions of traditional rendering techniques, in order to move beyond the standard triangle meshes and linear transformations that are ubiquitous in conventional computer graphics. Unlike purely theoretical works, we provide practical techniques in a form that lends itself to GPU-accelerated implementation, resulting in highly dynamic and interactive graphical representations. In particular, we focus primarily on GPU-based visualization techniques that utilize the programmability of the geometry and tessellation processor stages. With the aid of programmable shaders, we extend the capabilities of the conventional graphics pipeline to address the needs of *Information Visualization* systems.

By introducing generalized geometric primitives and transformations, which are computed on the GPU, we achieve several benefits. First, a wide range of visual representations can be obtained by varying parameters at different stages of the graphics pipeline. Second, these representations can be dynamically updated to allow for real-time interaction. Third, the changes can be interpolated to create fluid transitions. We believe the combination of these factors results in an attractive framework for *Information Visualization* in AR/VR environments. In particular, using our proposed techniques, it is possible to provide continuous feedback and allow for natural interactions when exploring multidimensional, multivariate data.

The individual sections of this article cover self-contained aspects of our approach towards visualization. While we were careful to ensure they can be used independently, when taken together, the sum is greater than the individual parts. In particular, our work re-frames structural theories of graphics with respect to the programmable graphics pipeline. By doing so, our article provides a conceptual framework and architecture for implementing GPU-accelerated visualizations. In this framework, a diverse set of common visualization types arises naturally from the underlying data through different configurations of *geometric primitives* and *coordinate transformations*. Starting from simple building blocks, it is possible to obtain complex visual representations, and apply trivial variations and extensions interactively as needed. Consequently, it enables viewing visualization types as part of a continuum, rather than a discrete catalog of options.

References

BAILEY, M. 2009. Using GPU shaders for visualization. *IEEE Computer Graphics and Applications*, 5, 96–100.

BAILEY, M. 2011. Using GPU shaders for visualization, part 2. *IEEE Computer Graphics and Applications 31*, 2 (March), 67–73.

BAILEY, M. 2013. Using GPU shaders for visualization, part 3. *IEEE Computer Graphics and Applications 33*, 3, 5–11.

BERTIN, J. 1983. *Semiology of Graphics: Diagrams, Networks, Maps (Berg W. J., Trans.)*. The University of Wisconsin Press, Madison, WI, USA.

BROSZ, J., NACENTA, M. A., PUSCH, R., CARPENDALE, S., AND HURTER, C. 2013. Transmogrification: causal manipulation of visualizations. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, ACM, 97–106.

CARD, S. K., MACKINLAY, J. D., AND SHNEIDERMAN, B. 1999. *Readings in information visualization: using vision to think*. Morgan Kaufmann.

CHU, A., FU, C.-W., HANSON, A. J., AND HENG, P.-A. 2009. GL4D: A gpu-based architecture for interactive 4d visualization. *Visualization and Computer Graphics, IEEE Transactions on 15*, 6, 1587–1594.

CLAESSEN, J. H., AND VAN WIJK, J. J. 2011. Flexible linked axes for multivariate data visualization. *Visualization and Computer Graphics, IEEE Transactions on 17*, 12, 2310–2316.

EVES, H. W. 1980. *Elementary matrix theory*. Dover Publications, Inc.

FISHER, R. A. 1936. The use of multiple measurements in taxonomic problems. *Annals of eugenics 7*, 2, 179–188.

FLOREK, M., AND NOVOTNỲ, M. 2006. Interactive information visualization using graphics hardware. In *Poster Proceedings of Spring Conference on Computer Graphics*, vol. 32.

GINSBURG, D., PURNOMO, B., SHREINER, D., AND MUNSHI, A. 2014. *OpenGL ES 3.0 programming guide*. Addison-Wesley Professional.

HECKBERT, P. 1994. *Graphics Gems IV*. Elsevier.

HEER, J., AND ROBERTSON, G. G. 2007. Animated transitions in statistical data graphics. *Visualization and Computer Graphics, IEEE Transactions on 13*, 6, 1240–1247.

HEER, J., AND SHNEIDERMAN, B. 2012. Interactive dynamics for visual analysis. *Queue 10*, 2, 30.

KHRONOS GROUP, 2015. OpenGL core specification 4.5. `http://www.opengl.org/registry/doc/glspec45.core.pdf`.

KOSARA, R., HAUSER, H., AND GRESH, D. L. 2003. An interaction view on information visualization. *State-of-the-Art Report. Proceedings of EUROGRAPHICS*.

LIU, Z., AND HEER, J. 2014. The effects of interactive latency on exploratory visual analysis. *IEEE transactions on visualization and computer graphics 20*, 12, 2122–2131.

MACKINLAY, J. D. 1986. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics (TOG) 5*, 2, 110–141.

MCCORMICK, B. H., DEFANTI, T. A., AND BROWN, M. D., 1987. Visualization in scientific computing.

MCDONNEL, B., AND ELMQVIST, N. 2009. Towards utilizing gpus in information visualization: A model and implementation of image-space operations. *Visualization and Computer Graphics, IEEE Transactions on 15*, 6, 1105–1112.

RUCHIKACHORN, P., AND MUELLER, K. 2015. Learning visualizations by analogy: Promoting visual literacy through visualization morphing. *Visualization and Computer Graphics, IEEE Transactions on 21*, 9, 1028–1044.

SALOMON, D. 2012. *Computer graphics and geometric modeling*. Springer Science & Business Media.

SATYANARAYAN, A., AND HEER, J. 2014. Lyra: An interactive visualization design environment. *Computer Graphics Forum (Proc. EuroVis)*.

STOLTE, C., TANG, D., AND HANRAHAN, P. 2002. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *Visualization and Computer Graphics, IEEE Transactions on 8*, 1, 52–65.

WEISKOPF, D. 2007. *GPU-based interactive visualization techniques*. Springer.

WICKHAM, H. 2010. A layered grammar of graphics. *Journal of Computational and Graphical Statistics 19*, 1, 3–28.

WILKINSON, L. 2005. *The grammar of graphics*. Springer.

WONG, P. C., AND BERGERON, R. D. 1994. 30 years of multidimensional multivariate visualization. In *Scientific Visualization*, 3–33.